Electronic Notes in Theoretical Computer Science 48 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume48.html pp. 1 – 20

On the Number of Rule Applications in Constraint Programs

Thom Frühwirth [1]

Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstrasse 67, D-80538 Munich, Germany

Abstract

We predict the maximal number of rule applications, i.e. worst-case derivation lengths of computations, in rule-based constraint solver programs written in the CHR language. CHR are a committed-choice concurrent constraint logic programming language consisting of multi-headed guarded rules. The derivation lengths are derived from rankings used in termination proofs for the respective programs. We are especially interested in rankings that give us a good upper bound, we call such rankings *tight*. Based on test-runs with randomized data, we compare our predictions with empirical results by considering constraint solvers ranging from Boolean and terminological constraints to arc-consistency and path-consistency.

Key words: Program Analysis, Termination, Derivation Lengths, Constraint Solving, Constraint Handling Rules.

1 Introduction

CHR (Constraint Handling Rules) [7] are a committed-choice concurrent constraint logic programming language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. CHR define both simplification of and propagation over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification.

In [9], to prove termination of CHR programs, we use a ranking that maps head and body of each rule in a CHR program to natural numbers, such that the rank of the head is strictly larger than the rank of the body. Intuitively then, the rank of a query yields an upper bound on the number of rule applications (derivation steps), i.e. derivation lengths.

[1] Email: fruehwir@informatik.uni-muenchen.de

Based on test-runs with randomized data, we compare our predictions with empirical results by considering constraint solvers ranging from Boolean and terminological to arc-consistent and path-consistent constraints.

Example 1.1 Consider the constraint `even` that ensures that a positive natural number in successor notation is even:

```
even(0) <=> true.
even(s(N)) <=> N=s(M),even(M).
```

The first rule says that `even(0)` can be simplified to `true`, a built-in constraint that is always true. In the second rule, the built-in constraint `=` stands for syntactical equality: `N=s(M)` ensures that `N` is the successor of some number `M`. The rule says that if the argument of `even` is the successor of some number `N`, then the predecessor of this number `M` must be even in order to ensure that the initial number `s(N)` is even.

If a constraint matches the head of a rule, it is replaced by the body of the rule. If no rule matches a constraint, the constraint delays. For example, the query `even(N)` delays. The query `even(0)` reduces to `true` with the first rule. To the query `even(s(N))` the second rule is applicable, the answer is `N=s(M),even(M)`. The query `even(s(0))` results in an inconsistency after application of the second rule, since `0=s(M)` is inconsistent.

An obvious ranking is

$rank(0) = 1$
$rank(s(N)) = 1 + rank(N)$

The ranking gives us an upper bound on the derivation length, since with each rule application, we decrease the rank of the argument of `even` by 2.

Related Work. To the best of our knowledge, there is no work in logic programming concerned with predicting derivation lengths for concrete programs. Somewhat related is [4], where an instance of quantative observables is used to prove termination of probabilistic CCP programs based on finite average derivation lengths. In the context of transforming CCP programs, where derivation length corresponds to the number of procedure expansions (unfolding steps), this measure is used to compare the efficiency of transformed programs in [3].

Overview of the Paper. This paper is a revised and extended version of [8]. The main extension concerns the empirical results which are presented here for the first time. We will first give syntax and semantics for CHR. Then, we introduce rankings and show how they can be used to derive tight upper bounds for worst-case derivation lengths. The main, fourth section reviews various CHR constraint solver programs and gives rankings for them. Based on the rankings, derivation lengths are discussed and empirical results from randomized test-runs of the constraint solvers are presented and evaluated. We conclude with a discussion of the results obtained.

2 Syntax and Semantics

In this section we give syntax and semantics for CHR, for details see [1]. We assume some familiarity with (concurrent) constraint (logic) programming [12,10,14].

A *constraint* is a predicate (atomic formula) in first-order logic. We distinguish between *built-in (or predefined) constraints* and *CHR (or user-defined) constraints*. Built-in constraints are those handled by a given constraint solver. CHR constraints are those defined by a CHR program.

In the following *abstract syntax*, upper case letters stand for conjunctions of constraints.

Definition 2.1 A CHR program is a finite set of CHR. There are two kinds of CHR. A *simplification CHR* is of the form

N @ H <=> $G \mid B$

and a *propagation CHR* is of the form

N @ H ==> $G \mid B$

where the rule has an optional name N followed by the symbol @. The multi-head H is a conjunction of CHR constraints. The optional guard G followed by the symbol | is a conjunction of built-in constraints. The body B is a conjunction of built-in and CHR constraints.

The *operational semantics* of CHR programs is given by a state transition system. With *derivation steps (transitions, reductions)* one can proceed from one state to the next. A *derivation* is a sequence of derivation steps.

Definition 2.2 A *state (or: goal)* is a conjunction of built-in and CHR constraints. An *initial state (or: query)* is an arbitrary state. In a *final state (or: answer)* either the built-in constraints are inconsistent or no derivation step is possible anymore.

Definition 2.3 Let P be a CHR program and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows. All upper case letters occurring in states stand for conjunctions of constraints.

Simplify
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge D$
if $(H$ <=> $G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$

Propagate
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge H' \wedge D$
if $(H$ ==> $G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$

When we use a rule from the program, we will rename its variables using new symbols, and these variables are denoted by the sequence $\bar{x}$. A rule with head H and guard G is *applicable* to CHR constraints H' in the context of

constraints D, when the condition holds that $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$. Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone.

If an applicable simplification rule (`H <=> G | B`) is applied to the CHR constraints H', the **Simplify** transition removes H' from the state, adds the body B to the state and also adds the equation $H = H'$ and the guard G. If a propagation rule (`H ==> G | B`) is applied to H', the **Propagate** transition adds B, $H = H'$ and G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints [1]. In this paper we are only concerned with simplification rules.

We finally discuss in more detail the rule applicability condition $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$. The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. More precisely, by $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$ we mean $(H_1 = H'_1) \wedge \ldots \wedge (H_n = H'_n)$, where conjuncts can be permuted. By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality and is usually implemented by a unification algorithm (as in Prolog).

Operationally, the rule applicability condition can be checked as follows: Given the built-in constraints of D, solve the built-in constraints $(H = H' \wedge G)$ without further constraining (touching) any variable in H' and D. This means that we first check that H' matches H and then check the guard G under this matching.

3 CHR Rankings

In this section, we introduce rankings and show how they can be used to derive tight upper bounds for worst-case derivation lengths.

3.1 Rankings

In [9] we prove termination for CHR programs under any scheduling of rule applications (independent from the search and selection rule). Roughly, a CHR program can be proven to terminate if we can prove that in each rule, the rank of the head is strictly larger than the rank of the body. We rely on polynomial interpretations, where the rank of a term or atom is defined by a linear positive combination of the rankings of its arguments.

Definition 3.1 Let f be a function or predicate symbol of arity n $(n \geq 0)$ and let $t_i (1 \leq i \leq n)$ be terms. A *CHR ranking* defines the rank of a term or atom $f(t_1, \ldots, t_n)$ as a natural number:

$$rank(f(t_1, \ldots, t_n)) = a_0^f + a_1^f * rank(t_1) + \ldots + a_n^f * rank(t_n)$$

where the a_i^f are natural numbers. For each variable X we impose $rank(X) \geq 0$.

This definition implies that $rank(t) \geq 0$ for all rankings in our scheme for all terms and atoms t. Instances of the ranking scheme *rank* specify the function and predicate symbols and the values of the coefficients a_i^f.

Example 3.2 The *(syntactic) size* of a term can be expressed in this scheme by:

$$size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots + size(t_n)$$

For example, the size of the term *f(a,g(b,c))* is 5. The size of *f(a,X)* is $2 + size(X)$ with $size(X) \geq 0$. This allows us to conclude that the term *f(g(X),X)* is larger in size than *f(a,X)* ($2 + 2 * size(X) \geq 2 + size(X)$), no matter what term X stands for.

A ranking for a CHR program will have to define the ranks of CHR and built-in constraints. We will define the rank of any built-in constraint to be 0, since we assume that they always terminate. A built-in constraint may imply order constraints between the ranks of its arguments (interargument relations), e.g. $s = t \rightarrow rank(s) = rank(t)$.

In extension of usual approaches, we have to define the rank of a conjunction of constraints, since CHR are multi-headed. The rank of a conjunction should reflect that conjunctions of CHR constraints are associative and commutative, but not idempotent. We define the rank of a conjunction as the sum of the ranks of its conjuncts:

$$rank((A \wedge B)) = rank(A) + rank(B)$$

In the following section, we will only give ranks for atomic CHR constraints, provided they are different from zero.

The following Theorem tells us how to prove CHR program termination.

Theorem 3.3 *[9] Given a CHR program P without propagation rules. Let the* CHR ranking condition of a simplification rule H `<=>` $G \mid B$ *be the formula*

$$\forall\ (OC_{G \wedge B} \rightarrow rank(H) > rank(B),$$

where $OC_{G \wedge B}$ is the conjunction of the order constraints implied by the built-in constraints in the guard and body of the rule. If the ranking condition holds for each rule in P, then P is terminating for all bounded goals. A goal G is bounded *if the rank of any instance of G is bounded from above by a constant k.*

3.2 Derivation Lengths from Rankings

The rank of a goal (query) gives us an upper bound on the number of rule applications (derivation steps), i.e. derivation lengths.

Theorem 3.4 *[8] Given a CHR program P without propagation rules. If the ranking condition holds for each rule in P, then the worst-case derivation length D_P for a bounded goal G in P is bounded by the rank of G. We write*

this as:

$$D_P \leq rank(G)$$

Proof. From the proof of Theorem 3.3 [9] we know that given an derivation step $G \longmapsto G_1$ it holds that $rank(G) > rank(G_1)$. Since ranks are natural numbers, we may rewrite this as $rank(G) \geq rank(G_1) + 1$. By induction we can show that given a derivation of length n, $G \longmapsto^n G_n$, we have that $rank(G) \geq rank(G_n)+n$. Since ranks are non-negative, this implies the desired $rank(G) \geq n$. □

We are interested in CHR rankings that get us as close as possible to the actual derivation lengths. This is the case if differences between the ranks of the heads and bodies of the rules in a program are bounded from above by a constant. We call such rankings *tight*.

Definition 3.5 Given a *CHR ranking of a simplification rule* H `<=>` $G \mid B$. The ranking is *exact for the rule* H `<=>` $G \mid B$ iff $rank(H) = rank(B) + 1$. The ranking is *tight by* n *for the rule* H `<=>` $G \mid B$ iff $rank(H) = rank(B)+n$, where n is a natural number. The ranking is *tight by* n *for a CHR program* P iff the ranking is tight by n_i for all rules in P and n is the maximum of all n_i.

The definition of tightness is appropriate for worst-case analysis, while average-case analysis would have to take into account the distribution of the n_i.

4 Derivation Lengths of Constraint Solvers

We now derive uppers bounds for the derivation lengths of actually implemented CHR constraint solvers. For each solver, we will give a ranking, we will relate the derivation length for a given goal to the number, c, of atomic constraints in the goal. and we give empirical results derived from test-runs with randomized data. We will summarize the results in a table, see e.g. Figure 1. The tables have the following columns:

Goal Gives the (abbreviated) goal that was run to produce the test data.

Worst Gives our predicted worst-case derivation length for the goal.

Apply Gives the actual number of rule applications, i.e. derivation length.

Try Gives the number of rules that have been tried, but not necessarily applied.

Time Gives the time to run the goal in seconds, including instrumented source code for randomization, on a Linux PC with medium work load. Only the relative size of the timings is of interest here.

The last two entries are given to show that the run time, i.e. time complexity, is more dependent on the number of rule trys than on the number of rule applications. There may be considerably more rule trys than applications.

The constraint solvers we discuss here (see also [7]) and the Prolog and CHR code that produced the test runs is available at
`www.informatik.uni-muenchen.de/~fruehwir/chr/complexity.pl`
Note that the CHR code under consideration in this paper has been written mainly for simplicity, not for efficiency. The code can be run via a WWW-interface on the internet using CHR online at the URL:
`www.pms.informatik.uni-muenchen.de/~webchr/`
We will use *concrete syntax* of Prolog-implementations of CHR, where a conjunction is a sequence of conjuncts separated by commas.

4.1 Boolean Algebra, Propositional Logic

The domain of Boolean constraints [15] includes the constants `0` for falsity, `1` for truth and the usual logical connectives of propositional logic, which are modeled here as CHR constraints. Syntactic equality `=` is a built-in constraint. In the constraint solver *Bool*, we simplify a single constraint into one or more equations whenever possible:

```
and(X,Y,Z) <=> X=0 | Z=0.
and(X,Y,Z) <=> Y=0 | Z=0.
and(X,Y,Z) <=> X=1 | Y=Z.
and(X,Y,Z) <=> Y=1 | X=Z.
and(X,Y,Z) <=> X=Y | Y=Z.
and(X,Y,Z) <=> Z=1 | X=1,Y=1.
```

For example, the first rule says that the constraint `and(X,Y,Z)`, when it is known that the first input argument `X` is 0, can be reduced to asserting that the output `Z` must be 0. Hence the goal `and(X,Y,Z),X=0` will result in `X=0, Z=0`.

Derivation Length. Since each CHR constraint is reduced to built-in constraints by a single rule application, the maximum derivation length is just the number, c, of constraints in the goal. Let the ranking be defined as

$rank(A) = 1$ if A is an atomic CHR constraint

For each rule in *Bool*, `H <=> G | B`, we have that $rank(H) = 1$ and $rank(B) = 0$. Hence the ranking is exact for all rules. Consequently, the worst-case derivation length of a Boolean goal is

$$D_{Bool} \leq c$$

It can be much smaller. For example, the goal `and(U,V,W)` delays, its derivation length is zero. Another example is a goal that contains the constraint `and(0,Y,1)`. If it is selected first, it will reduce to the inconsistent built-in constraint `1=0` in one derivation step. Because of the inconsistency, this is a final state of the derivation.

Goal	Worst	Apply	Try	Time
and(X,X,X), ..., and(Y,Y,Y)	8	8	40	0.01
test(500,A,B), A=1	500	1	3009	0.34
test(500,A,B), B=0	500	0	3006	0.35
test(500,A,B), A=0	500	500	3500	0.46
test(500,A,B), B=1	500	500	6000	0.73
A=0, test(500,A,B)	500	500	500	0.05
B=1, test(500,A,B)	500	500	6000	0.73

Fig. 1. Results from Test-Runs with Boolean And

As for the empirical results, consider Figure 1. The first entry in Figure 1 refers to the following goal in two variables `X` and `Y`

```
and(X,X,X), and(X,X,Y), and(X,Y,X), and(X,Y,Y),
and(Y,X,X), and(Y,X,Y), and(Y,Y,X), and(Y,Y,Y).
```

It will reduce to the constraint `X=Y` in 8 derivation steps. The Prolog predicate `test/3` produces a chain of `and` constraints, where the last variable of one constraint is the first variable of the next constraint. The first (A) and the last (B) variable are returned.

The table of Figure 1 shows that

- The actual derivation length ranges between 0 and the predicted worst case derivation length.
- The number of rule trys is up to 12 times larger than the worst-case derivation length. Note that there are 6 rules.
- Time is roughly proportional to the number of rule trys.
- The order of the (built-in) constraints may strongly influence the run time.

Boolean Cardinality

The cardinality constraint combinator was introduced in the CLP language cc(FD) [19] for finite domains. In the solver *Card* we adapted cardinality for Boolean variables. The Boolean cardinality constraint `#(L,U,BL,N)` is true if the number of Boolean variables in the list `BL` that are equal to `1` is between `L` and `U`. `N` is the length of the list `BL`. Boolean cardinality can express negation `#(0,0,[C],1)`, exclusive or `#(1,1,[C1,C2],2)`, conjunction `#(N,N,[C1,...,Cn],N)` and disjunction `#(1,N,[C1,...,Cn],N)`.

```
 % trivial, positive and negative satisfaction
triv_sat@ #(L,U,BL,N) <=> L=<0,N=<U | true.
pos_sat @ #(L,U,BL,N) <=> L=N | all(1,BL).
neg_sat @ #(L,U,BL,N) <=> U=0 | all(0,BL).
```

```
 % positive and negative reduction
pos_red @ #(L,U,BL,N) <=> delete(1,BL,BL1) |
                          0<U, #(L-1,U-1,BL1,N-1).
neg_red @ #(L,U,BL,N) <=> delete(0,BL,BL1) |
                          L<N, #(L,U,BL1,N-1).
```

In this CHR program, all constraints except cardinality are built-in. `all(B,L)` equates all elements of the list `L` to `B`. `delete(X,L,L1)` deletes the element `X` from the list `L` resulting in the list `L1`. Due to the semantics of guard evaluation, `X` must exactly match the element to be removed.

Derivation Length. Our ranking is based on the length of the list argument of `#`:

$$rank(\#(L,U,BL,N)) = 1 + length(BL)$$

$$length([]) = 0$$
$$length([X|L]) = 1 + length(L)$$

The rank adds one to the length of the list in order to give a cardinality with the empty list a positive rank. For example, consider the goal `#(0,0,[],0)`. Any of the three satisfaction rules can be applied to it and the derivation length will always be one.

From the ranking we see that the derivation length of a single cardinality constraint is bounded by the length of the list argument. For example, the goal `#(1,1,[0,0,0,0,X],5)` needs five derivation steps to reduce to `X=1`. The first four steps remove the zeros from the list. The derivation length of a goal is less or equal to the sum of the lengths of the lists occurring in the goal. Hence it is linear in the syntactic size of the goal in the worst case.

If the maximum length of the lists is bounded by $l - 1$, we have that:

$$D_{Card} \leq c * l$$

The ranking is exact for the two recursive reduction rules, because of the order constraint implied by `delete`. It is tight by l only for the three satisfaction rules, since a cardinality constraint with arbitrary rank may be reduced to built-in constraints with rank 0 in one derivation step. Hence the solver program *Card* is tight by l.

Our empirical results are presented in Figure 2. The list L has length 500. *allr* is a variation on *all*. *allb* produces a list of alternating zeros and ones. *card_random* produces a random list of free variables, zeros and ones. *random* produces a random number inside a given range. *rand_range* produces a random range. The table shows that

- The actual derivation length ranges between 0 and the predicted worst case derivation length. On average, it is about half of the predicted length.
- The number of rule trys is up to 5 times larger than the worst-case derivation length. Note that there are 5 rules. On average, it is about two times larger

Goal	Worst	Apply	Try	Time
#(0,0,[],0)	1	1	1	0.0
#(1,1,[0,0,0,0,X],5)	6	5	22	0.0
all(0,L), #(1,1,[X|L],N+1)	502	0	5	0.0
#(1,1,[X|L],N+1), all(0,L)	502	0	2505	5.13
#(1,1,[X|L],N+1), allr(0,L)	502	0	2505	5.22
all(Y,L), #(1,1,[X|L],N+1), Y=0	502	0	10	0.02
card_random(500,A,B,L), #(A,B,L,500)	501	330	1500	2.95
card_random(500,A,B,L), #(A,B,L,500)	501	199	828	1.49
card_random(500,A,B,L), #(A,B,L,500)	501	339	1527	3.05
card_random(500,A,B,L), #(A,B,L,500)	501	327	1476	2.98
card_random(500,A,B,L), #(A,B,L,500)	501	318	1434	2.88
random(0,500,A),allb(L),#(A,A,L,500)	501	109	435	0.46
random(0,500,A),allb(L),#(A,A,L,500)	501	434	1917	3.50
random(0,500,A),allb(L),#(A,A,L,500)	501	370	1597	2.79
random(0,500,A),allb(L),#(A,A,L,500)	501	200	799	1.05
random(0,500,A),allb(L),#(A,A,L,500)	501	120	479	0.52
rand_range(500,A,B),allb(L),#(A,B,L,500)	501	337	1431	2.45
rand_range(500,A,B),allb(L),#(A,B,L,500)	501	262	1056	1.57
rand_range(500,A,B),allb(L),#(A,B,L,500)	501	410	1796	3.26
rand_range(500,A,B),allb(L),#(A,B,L,500)	501	106	423	0.45
rand_range(500,A,B),allb(L),#(A,B,L,500)	501	422	1856	3.41

Fig. 2. Results from Test-Runs with Boolean Cardinality

than the worst case and about four times larger than the actual number of rule applications.

- Time is roughly proportional to the number of rule trys.
- The order of the (built-in) constraints may strongly influence the run time.

4.2 Path Consistency

In this section we analyze constraint solvers that implement the classical artificial intelligence algorithm of path consistency [13,16]. We use abstract syntax in the following definitions.

Definition 4.1 A *disjunctive binary constraint* c_{xy}, X $\{r_1, \ldots, r_n\}$ Y, is a

finite disjunction $(X\ r_1\ Y) \vee \ldots \vee (X\ r_n\ Y)$, where each r_i is a binary relation. The r_i are called *primitive constraints*.

A *binary constraint network* is a conjunction of disjunctive binary constraints. The network can be represented by a *directed constraint graph*, where the nodes denote variables and the arcs are labeled by binary constraints.

Usually, the number p of primitive constraints is finite and they are pairwise disjoint. We will assume so in the following.

For example, $A\ \{<\}\ B, A\ \{<,>\}\ B, A\ \{<,=,>\}\ B$ are disjunctive binary constraints c_{AB} between A and B. $A\ \{<\}\ B$ means $A < B$, and $A\ \{<,>\}\ B$ means $A \neq B$. Finally, $A\ \{<,=,>\}\ B$ is always true.

Definition 4.2 A network is *path consistent* if for pairs of nodes (i, j) and all paths $i - i_1 - i_2 \ldots i_n - j$ between them, the direct constraint c_{ij} is at least as tight as the indirect constraint along the path, i.e. the composition of constraints $c_{ii_1} \otimes \ldots \otimes c_{i_nj}$ along the path.

It follows from the definition of path consistency that we can intersect the direct and indirect constraint to arrive at a tighter direct constraint. Let intersection be denoted by the operator $\oplus$. A graph is *complete* if there is a pair of arcs, one in each direction, between every pair of nodes. If the graph underlying the network is complete it suffices to consider paths of length 2 at most: For each triple of nodes (i, k, j) we repeatedly compute $c_{ij} := c_{ij} \oplus (c_{ik} \otimes c_{kj})$ until a fixpoint is reached. This is the basic path consistency algorithm.

For example, given $I \leq K\ \wedge\ K \leq J\ \wedge\ I \geq J$ and taking the triple (i, k, j), $c_{ik} \otimes c_{kj}$ results in $I \leq J$ and the result of intersecting it with c_{ij} is $I = J$. From (j, i, k) we get $J = K$ (we can compute c_{ji} from c_{ij}). From (k, j, i) we get $K = I$. Another round of computation causes no more change, so the fixpoint is reached with $I = J\ \wedge\ J = K\ \wedge\ K = I$.

Let the disjunctive binary constraint c_{ij} be represented in concrete syntax by the CHR constraint `c(I,J,R)` where `I` and `J` are the variables and `R` is its set of primitive constraints. The basic operation of path consistency, $c_{ij} := c_{ij} \oplus (c_{ik} \otimes c_{kj})$, can be implemented directly by the rule:

```
path_consistency @
c(I,K,R1), c(K,J,R2), c(I,J,R3) <=>
      composition(R1,R2,R12),intersection(R3,R12,R4),
      R3<>R4 |
      c(I,K,R1), c(K,J,R2), c(I,J,R4).
```

In this solver *Path*, the operations $\otimes$ and $\oplus$ are implemented by the built-in constraints `composition` and `intersection`. Composition of disjunctive constraints can be computed by pairwise composition of its primitive constraints. Intersection for disjunctive constraints can be implemented by set intersection. In the guard of the rule, the check `R3<>R4` makes sure that the new constraint `R4` is different from the old one `R3`. Instances of a similar solver have been

used for temporal reasoning [6] and for spatial reasoning [5].

Derivation Length. We rely on the following ranking:

$rank(c(I, K, C)) = card(C)$

$card(\{r_1, \ldots, r_n\}) = n$

Every rule application removes at least one primitive constraint and at most all of them from the set of primitive constraints `R3` by intersecting it with `R12`. If the maximum number of primitive constraints is p, the ranking is tight by at most p. The actual tightness depends on the intersection behavior of the set of primitive constraints.

For the derivation lengths we have that:

$$D_{Path} \leq c * p$$

i.e. the worst-case derivation length is linear in the syntactic size of the goal.

Goal	Worst	Apply	Try	Time
length(L,5),tpath(L,A)	60	11	201	0.14
length(L,5),tpath(L,A)	60	9	152	0.11
length(L,5),tpath(L,A)	60	12	195	0.14
length(L,5),tpath(L,A)	60	10	187	0.14
length(L,5),tpath(L,A)	60	8	137	0.11
length(L,10),tpath(L,A)	270	64	2622	1.92
length(L,10),tpath(L,A)	270	68	2437	1.80
length(L,10),tpath(L,A)	270	72	2878	2.10
length(L,10),tpath(L,A)	270	81	3041	2.26
length(L,10),tpath(L,A)	270	61	2275	1.66
length(L,20),tpath(L,A)	1140	261	23514	21.41
length(L,20),tpath(L,A)	1140	251	23265	21.33
length(L,20),tpath(L,A)	1140	248	23760	21.48

Fig. 3. Results from Test-Runs with Path Consistency

In the goals of Figure 3, *tpath* generates a pair of c constraints between each pair of different variables in its argument list. The disjunctive constraint for c is a randomly choosen non-empty subset of $\{<, =, >\}$, hence $p = 3$. (as in the earlier examples of this section). For a list of length n, there are exactly $n * (n - 1)$ constraints. Thus three times as much is the worst case derivation length. The table shows that

- The behavior of the random problem instances is quite stable.
- The actual derivation length is proportional to the predicted worst case

derivation length, it is about a quarter. It is less than the number of constraints in our examples.

- The number of rule trys increases faster than the worst-case derivation length. It is roughly cubic in the number of variables, while the derivation length is quadratic.
- Time is roughly proportional to the number of rule trys.

Adding up the individual ranks of each constraint would result in a more precise worst-case estimate of the derivation length.

4.3 Interval Constraints, Arc Consistency

The following rules of the solver *Intv* implement an arc consistency algorithm for interval constraints (a special case of finite domain constraints) [18,2]. The main idea of arc consistency is that it distinguishes a special class of unary constraints of the form $X \in D$, where D is a finite set of given values.

Definition 4.3 A conjunction of unary constraints $X_1 \in D_1 \wedge \ldots \wedge X_n \in D_n$ is *arc consistent* with respect to a constraint $c(X_1, \ldots, X_n)$, if for all $i \in \{1, \ldots, n\}$ and for all possible values for X_i taken from its domain D_i the constraint $X_1 \in D_1 \ \wedge \ldots \wedge \ X_n \in D_n \ \wedge \ c(X_1, \ldots, X_n)$ is satisfiable.

In other words, in an arc consistent conjunction of constraints, every value of every domain takes part in a solution. A conjunction of constraints can be made arc consistent by deleting those values from the domain of the variables that do not participate in any solution of the constraints.

In our case, the domains are intervals of integers, and values are deleted from domains by making intervals smaller. The unary interval constraint `X in A:B` stands for $X \in \{n \in Int \mid A \leq n \wedge n \leq B\}$. `in, le, eq` and `add` are CHR constraints, the inequalities `<, =<, >, >=, <>` are built-in arithmetic constraints, and `min, max, +, -` are built-in arithmetic functions. Intervals of integers are closed under computations involving only these functions. The built-in prefix operator `not` negates its argument.

```
 % Interval Constraints
inconsistency @ X in A:B <=> A>B | false.
intersection  @ X in A:B, X in C:D <=> A=<B,C=<D |
     X in max(A,C):min(B,D).

 % (In)equalities
le @ X le Y, X in A:B, Y in C:D <=> A=<B,C=<D, B>D |
     X le Y, X in A:D, Y in C:D.
le @ X le Y, X in A:B, Y in C:D <=> A=<B,C=<D, C<A |
     X le Y, X in A:B, Y in A:D.

eq @ X eq Y, X in A:B, Y in C:D <=> A=<B,C=<D, A<>C |
     X eq Y, X in max(A,C):B, Y in max(C,A):D.
```

```
eq @ X eq Y, X in A:B, Y in C:D <=> A=<B,C=<D, B<>D |
     X eq Y, X in A:min(B,D), Y in C:min(D,B).

 % Addition X+Y=Z
add @ add(X,Y,Z), X in A:B, Y in C:D, Z in E:F <=>
   A=<B,C=<D,
   not (A>=E-D,B=<F-C,C>=E-B,D=<F-A,E>=A+C,F=<B+D) |
     add(X,Y,Z),
     X in max(A,E-D):min(B,F-C),
     Y in max(C,E-B):min(D,F-A),
     Z in max(E,A+C):min(F,B+D).
```

The rules affect the interval constraints only, the constraints `le, eq` and `add` remain unaffected. The rules `inconsistency` and `intersection` remove one interval constraint each. The built-in inequalities `A=<B` and `C=<D` used in the guards of the rules ensure that these rules apply only to non-empty intervals. The remaining built-in inequalities in the guards ensure that in each rule, at least one interval gets strictly smaller.

Derivation Length. We rank constraints by the width (size) of their intervals:

$$rank(X \ in \ A : B) = 2 + width(A : B)$$

$$width(A : B) = B - A \text{ if } A \leq B$$
$$width(A : B) = -1 \text{ otherwise}$$

For the ranking, 2 is added to the interval width such that empty and singleton intervals have positive ranks as well.

Let w be the the maximum rank of an interval constraint in a given goal. The tightness of a rule can be computed by assuming that all interval constraints have maximum rank w except those whose intervals are computed in the body, they have minimum rank 1. The `inconsistency` rule is exact. For the remaining rules we have that $w > 1$. The `intersection` rule is tight by $2w - 1$, the rules for `eq` and `le` are tight by $w - 1$, the rule for `add` is tight by $3w - 3$. Hence the solver program *Intv* is exact for $w = 1$ and tight by $3w - 3$ for $w > 1$. The derivation length is bounded by the sum of the interval sizes in a goal:

$$D_{Intv} \leq c * w$$

Assume we drop the rule `add` from the solver. Then the interval computations use only `min` and `max`, i.e. no new numbers can be computed for the interval bounds. Let there be n different numbers in the intervals of the goal. Then we can replace the maximal interval constraint rank w by the tighter n.

In Figure 4 *tadd* takes a list of n different variables and produces the constraints *add(A,B,C), A le C* between three subsequent variables for every other variable. So for n variables, exactly $n - 2$ constraints are produced. The interval domains for the variables are generated randomly, they are non-negative and the upper bound increases by 100 for every other variable to

Goal	Worst	Apply	Try	Time
add(A,B,C), add(C,A,B), A in 0:7,...	24	8	34	0.02
tadd([A,B,C,D,E,F],100)	1208	15	72	0.02
tadd([A,B,...,P],100)	9776	36	183	0.05
tadd([A,B,...,P],100)	9776	33	174	0.05
tadd([A,B,...,P],100)	9776	32	171	0.04
tadd([A,B,...,P],100)	9776	40	204	0.06
len(L,100),tadd(L,100)	490196	352	1894	0.51
len(L,100),tadd(L,100)	490196	352	1894	0.50
len(L,100),tadd(L,100)	490196	340	1843	0.49
len(L,100),tadd(L,100)	490196	339	1831	0.50
len(L,100),tadd(L,100)	490196	349	1885	0.51
len(L,200),tadd(L,100)	1980396	718	3869	1.04
len(L,200),tadd(L,100)	1980396	702	3794	1.02
len(L,200),tadd(L,100)	1980396	706	3809	1.03
len(L,200),tadd(L,100)	1980396	715	3854	1.06
len(L,200),tadd(L,100)	1980396	714	3848	1.03
len(L,10),U in 1:1,genless(U,L,Z),...	2040	884	3737	1.11
len(L,20),U in 1:1,genless(U,L,Z),...	4080	1420	7243	2.12
len(L,30),U in 1:1,genless(U,L,Z),...	6120	2308	13569	3.96
len(L,40),U in 1:1,genless(U,L,Z),...	8160	3735	24973	7.23
len(L,50),U in 1:1,genless(U,L,Z),...	10200	5482	40967	11.78
len(L,60),U in 1:1,genless(U,L,Z),...	12240	7549	62251	17.87

Fig. 4. Results from Test-Runs with Interval Arc Consistency

increase the probability of consistency in presence of the constraint *A le C*. Hence the maximum interval domain size is $2 + 50n$. *genless* generates a sequence of n finally inconsistent *add* constraints involving n variables, all domains have width 202. The table shows that

- The behavior of the random problem instances is quite stable.
- The actual derivation length is usually much better than the predicted worst case derivation length, but the last entries shows that depending on the problem type, the worst case can be eventually reached as problem size increases.

- The number of rule trys is roughly proportional to the number of rule applications, except for the goals involving *genless*.
- Time is roughly proportional to the number of rule trys.

4.4 Terminological Reasoning, Description Logic

Terminological formalisms (aka description logics) [17] are used to represent the terminological knowledge of a particular problem domain on an abstract logical level. One starts with atomic concepts and roles, and then defines new concepts and their relationship in terms of existing concepts and roles. Concepts can be considered as unary relations similar to types. Roles correspond to binary relations over objects. In this paper, we use a natural language like syntax to help readers not familiar with the formalism.

Definition 4.4 *Concept terms* are defined inductively: Every *concept (name)* c is a concept term. If s and t are concept terms and r is a *role (name)*, then the following expressions are also concept terms:

s `and` t (conjunction), s `or` t (disjunction), `nota` s (complement),
`every` r `is` s (value restriction), `some` r `is` s (exists-in restriction).

Objects are constants or variables. Let a, b be objects. Then $a : s$ is a *membership assertion* and $(a, b) : r$ is a *role-filler assertion*. An *A-box* is a conjunction of membership and role-filler assertions.

Definition 4.5 A *terminology (T-box)* consists of a finite set of acyclic *concept definitions*

c `isa` s,

where c is a newly introduced concept name and s is a concept term.

The CHR constraint solver *Descr* for description logics is similar to the one in [11], except that here we represent both the A-box and the T-box as constraints. The solver simplifies and propagates assertions in the A-box by using the definitions in the T-box. It makes information more explicit and looks for obvious contradictions such as `X : man` and `X : nota man`. This is handled by the rule:

```
I : nota S, I : S  <=> false.
```

The unfolding rules replace concept names by their definitions.

```
I : C, C isa S  <=>  I : S, C isa S.
I : nota C, C isa S  <=>  I : nota S, C isa S.
```

The conjunction rule generates two new, smaller assertions:

```
I : S and T  <=>  I : S, I : T.
```

Disjunction is handled by lazy search, not directly by CHR. An exists-in re-

striction generates a new variable that serves as a "witness" for the restriction:

```
I : some R is S  <=>  (I,J) : R, J : S.
```

A value restriction has to be propagated to all role fillers using a propagation rule:

```
I : every R is S, (I,J) : R  ==>  J : S.
```

The final simplification rules push the complement operator `nota` down to the leaves of a concept term:

```
I : nota nota S  <=>  I : S.
I : nota (S or T)  <=>  I : nota S and nota T.
I : nota (S and T)  <=>  I : nota S or nota T.
I : nota (every R is S)  <=>  I : some R is nota S.
I : nota (some R is S)  <=>  I : every R is nota S.
```

Note that the only CHR constraints that are rewritten by the rules are membership assertions.

Derivation Length. We rank constraints by the size of their concept terms:

$rank(I : s) = size(s)$

$size(nota\ s) = 2 * size(s)$
$size(some\ r\ is\ s) = 1 + size(s)$
$size(every\ r\ is\ s) = 1 + size(s)$
$size(c) = 1 + size(s)$ if (`c isa s`) exists
$size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots + size(t_n)$ otherwise.

The derivation length D_{Descr} is bounded by the sum of the sizes of the concept terms occurring in a goal. Since the size of a concept depends on its definition, the syntactic size of the goal does not properly reflect the worst-case derivation length. Let the maximum size of a concept term be bounded by a constant k.

The ranking is exact for all but three rules: the rule involving complement and concept definition, which is tight by 2, the rule handling contradiction (tight by at most $3k/2$) and the rule for double complement (tight by at most $3k/4$).

As long as search for disjunction and the propagation rule for value restrictions is not involved, we have that

$$D_{Descr'} \leq c * k$$

Search and value restriction give rise to exponential time-complexity [8].

In Figure 5, *gen_dl* randomly generates a concept term of a given depth. Each kind of concept forming operator (*nota, and, ..., some*) has the same probability. The worst case derivation length is the size of the concept term T. The table shows that

- The actual derivation length is between one and the predicted worst case derivation length minus one.

Goal	Worst	Apply	Try	Time
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	152	1	1	0.0
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	156	29	29	0.02
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	208	45	45	0.02
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	240	16	16	0.01
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	190	1	1	0.0
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	386	246	246	0.14
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	254	132	132	0.08
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	114	113	113	0.05
gen_dl(1,T,10,A), I::T,(I,J)::r,(I,K)::r	228	16	16	0.01

Fig. 5. Results from Test-Runs with Description Logic Constraints

- The number of rule trys is identical to the number of rule applications, due to the simple structure of the rules.
- Time is roughly proportional to the number of rule trys.

5 Conclusions

We predicted the maximal number of rule applications, i.e. worst-case derivation lengths of computations, in CHR constraint solver programs. The derivation lengths are derived from tight rankings used in termination proofs. Usually, the worst-case derivation length D is linear in the size of the goal. D is bounded by $c * r$, where c is the number of atomic constraints in the goal and r is the maximum rank of an atomic constraint. Except for the terminological solver *Descr* and the interval solver *Intv*, the syntactic size of a goal properly reflects its worst-case derivation length.

Our empirical results show that

- The predicted worst case derivation length can be reached in practice. The average derivation length is typically proportional to the worst case length, except for interval arc consistency.
- The number of rule trys is at least linear in the number of rule applications, but it may increase much faster.
- Time is roughly proportional to the number of rule trys, typically not to the number of rule applications.

These results show that the derivation length does not necessarily reflect the time complexity of a CHR program. The main reason is that the number of rule applications does not take into account the effort of finding the appropriate combination of constraints in the goal that match the multi-head of a

rule. This effort is reflected in the number of rule trys.

While the empirical results have shown the precision of our prediction for the worst case number of rule applications is tight, future work should be concerned with average case analysis and with predicting the time complexity of a CHR program from its rules.

References

[1] Abdennadher, S., *Operational semantics and confluence of constraint propagation rules*, in: *3rd Intl. Conf. on Principles and Practice of Constraint Programming*, LNCS **1330** (1997), pp. 252–266.

[2] Benhamou, F., *Interval constraint logic programming*, in: A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS **910**, Springer-Verlag, 1995 pp. 1–21.

[3] Bertolino, M., S. Etalle and C. Palamidessi, *The replacement operation for ccp programs*, in: A. Bossi, editor, *Proceedings of 9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS **1817** (2000), pp. 216–233.

[4] Di Pierro, A. and H. Wiklicky, *Quantitative observables and averages in probabilistic constraint programming*, in: K. Apt, A. Kakas, E. Monfroy and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop*, Springer, Berlin, Heidelberg, 2000 pp. 212–236.

[5] Escrig, M. and F. Toledo, "Qualitative Spatial Reasoning: Theory and Practice," IOS Press, 1998.

[6] Frühwirth, T., *Temporal reasoning with constraint simplification rules*, Technical Report ECRC-94-05, ECRC, Munich (1994).

[7] Frühwirth, T., *Theory and practice of constraint handling rules, special issue on constraint logic programming*, Journal of Logic Programming **37** (1998), pp. 95–138.

[8] Frühwirth, T., *Predicting derivation lengths in rule-based constraint programs*, in: *Neuviëmes Journëes Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'2000)*, Marseille, France, 2000.

[9] Frühwirth, T., *Proving termination of constraint solver programs*, in: K. Apt, A. Kakas, E. Monfroy and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop*, Springer, Berlin, Heidelberg, 2000 .

[10] Frühwirth, T. and S. Abdennadher, "Constraint-Programming," Springer, Berlin, 1997.

[11] Frühwirth, T. and P. Hanschke, *Terminological reasoning with constraint handling rules*, in: P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming* (1995), pp. 361–381.

[12] Jaffar, J. and M. J. Maher, *Constraint logic programming: A survey*, The Journal of Logic Programming **19 & 20** (1994), pp. 503–581.

[13] Mackworth, A. K. and E. C. Freuder, *The complexity of some polynomial network consistency algorithms for constraint satisfaction problems*, Artificial Intelligence **25** (1985), pp. 65–73.

[14] Marriott, K. and P. J. Stuckey, "Programming with Constraints: An Introduction," MIT Press, 1998.

[15] Menju, S., K. Sakai, Y. Sato and A. Aiba, *A study on boolean constraint solvers*, in: F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, MIT Press, London, 1993 pp. 253–268.

[16] Mohr, R. and T. Henderson, *Arc and Path Consistency Revisited*, Artificial Intelligence **28** (1986), pp. 225–233.

[17] Patel-Schneider, P. and M.-C. Rousset, *Special issue on description logics* (1999).

[18] van Hentenryck, P., Y. Deville and C.-M. Teng, *A generic arc-consistency algorithm and its specializations*, Artificial Intelligence **57** (1992), pp. 291–321.

[19] van Hentenryck, P., V. Saraswat and Y. Deville, *Constraint processing in cc(fd)*, in: A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS **910**, Springer-Verlag, 1995 pp. 1–21.